A Sampling Method Focusing on Practicality

Daniel Gracia-Perez, Hugues Berry, Olivier Temam

A Sampling Method Focusing on Practicality *

Daniel Gracia Pérez
daniel.gracia-perez@cea.fr
CEA List, France

Hugues Berry, Olivier Temam
{hugues.berry,olivier.temam}@inria.fr
INRIA Futurs, France

Abstract. In the past few years, several research works have demonstrated that sampling can drastically speed up architecture simulation, and several of these sampling techniques are already largely used. However, for a sampling technique to be both easily and properly used, i.e., plugged and reliably used into many simulators with little or no effort or knowledge from the user, it must fulfill a number of conditions: it should require no hardware-dependent modification of the functional or timing simulator, it should simultaneously consider warm-up and sampling, while still delivering high speed and accuracy.

The motivation for this article is that, with the advent of generic and modular simulation frameworks like ASIM, SystemC, LSE, MicroLib or UniSim, there is a need for sampling techniques with the aforementioned properties, i.e., which are almost entirely *transparent* to the user and simulator agnostic. In this article, we propose a sampling technique focused more on transparency than on speed and accuracy, though the technique delivers almost state-of-the-art performance. Our sampling technique is a hardware-independent and integrated approach to warm-up and sampling; it requires no modification of the functional simulator and solely relies on the performance simulator for warm-up. We make the following contributions: (1) a technique for splitting the execution trace into a potentially very large number of variable-size regions to capture program dynamic control flow, (2) a clustering method capable of efficiently coping with such a large number of regions, (3) a *budget*-based method for jointly considering warm-up and sampling costs, presenting them as a single parameter to the user, and for distributing the number of simulated instructions between warm-up and sampling based on the region partitioning and clustering information.

Overall, the method achieves an accuracy/time tradeoff that is close to the best reported results using clustering-based sampling (though usually with perfect or hardware-dependent warm-up), with an average CPI error of 1.68% and an average number of simulated instructions of 288 million instructions over the Spec benchmarks. The technique/tool can be readily applied to a wide range of benchmarks, architectures and simulators, and will be used as a sampling option of the UniSim modular simulation framework.

1 Introduction and Related Work

Sampling is a delicate *balance* or *tradeoff* between simulation accuracy, overall simulation time, and practicality (scope of target architectures, user effort, transparency for the user). All these characteristics are important for a sampling technique to be efficient and useful for architecture researchers. SimPoint [20] can be credited for sparking a surge of interest in sampling because the technique is both efficient and easy to use. Considering the achievements of SimPoint and of later techniques/improvements [25, 12, 10, 24], do we need to push sampling research any further? Most sampling techniques have been applied to a specific simulator. However, with the advent of modular simulation frameworks such as ASIM [6], SystemC [21], the Liberty Simulation Environment (LSE) [23], MicroLib [19] or UniSim [22], there is a need for a sampling technique which is as independent as possible of the simulator/target architecture. In such frameworks, the simulation is driven by a generic engine which calls the different simulator modules, so any sampling technique would have to be plugged directly in the engine itself. But plugging a sampling technique within the engine would also have the benefit of automatically providing the sampling capability to almost any simulator written for that framework. However, since the engine can be used to simulate a large range of architectures, the sampling technique must be as architecture-independent as possible, so the user can transparently use this capability. Note also that modular simulators are typically 10 to 20 times slower than monolithic simulators like SimpleScalar, so that sampling is critical for them. After surveying existing sampling techniques, we concluded that, in spite of significant recent progress, they still do not achieve a satisfactory accuracy/time/practicality tradeoff for end-users. We feel it is important to explain why in details, in order to better expose the rationale and con-

*This article is a modified version of the article originally published at IEEE Micro. In the IEEE Micro version, we compared our clustering technique against the technique used in SimPoint 2.0; in this version, we compare against SimPoint 3.0, where the speed of clustering was largely improved.

tributions of this article. Let us first characterize what a good accuracy/time/practicality tradeoff should be.

We consider a good *accuracy* target is of the order of one or a few percent, because designing an architecture mechanism is a trial-and-error process composed of many "micro decisions" (parameter values selection, choosing against two architecture options, etc...) based on simulation results which often correspond to small performance differences.

With respect to *time*, the functional simulation time largely dominates the overall simulation time (timing simulation of sampling intervals plus functional simulation between sampling intervals) because the sampling intervals only correspond to a few percent of the total number of instructions in the program trace, so improving sampling efficiency (reducing the total sample size) would bring little improvements in that context. However, recent studies, such as TurboSMARTS [24], SimPoint with LVS (Load Value Sequence) and MHS (Memory Hierarchy State) [3], show that checkpointing can drastically reduce overall simulation time of sampling techniques (from a few hours to a few minutes per benchmark) by getting rid of functional simulation. Considering the speedup factor brought by checkpointing, and considering the advent of more complex processor architectures (more complex superscalar processors, multi-cores) and slower modular simulation, checkpointing will likely become a necessary complement to sampling techniques in the near future. Assuming checkpointing will become a common feature, overall simulation time becomes again entirely determined by the *number of simulated instructions*. In other words, reducing the sampling size by X% reduces the overall simulation time by X% as well. Since this article is solely concerned with the sampling technique (not functional simulation vs. checkpointing), and since the proposed sampling technique is perfectly compatible with a checkpointing technique, throughout the article, *time* will denote the number of simulated instructions (as opposed to overall simulation time), and we will thus focus on reducing the total number of simulated instructions.

Finally, *practicality* may be the least measurable characteristic of the sampling tradeoff, but one should not forget that, in the end, scope and ease of use are just as important as efficiency for users to adopt a new methodology. Considering architecture researchers have been using the crude approach of randomly picking traces of arbitrary sizes for a long time, it is a safe bet to assume they will discard any technique that is not simple enough to use or which imposes too many restrictions. Among the three components of the sampling tradeoff (accuracy, time and practicality), we believe that the least attention has been paid to practicality. The rationale of this article is to achieve a good accuracy/time tradeoff without degrading practicality. We explain below why recent sampling techniques still have significant practicality limitations, especially when it comes to warm-up.

Why current sampling techniques are not yet satisfactory, from a practicality perspective. Arguably, the current best sampling techniques are SimPoint [20, 15, 16, 11, 10, 3], SMARTS/TurboSMARTS [25, 24], and EXPERT [12]. There are two possible approaches for selecting sampling intervals: either (1) pick a large number of uniformly (or randomly) selected small intervals, or (2) pick a few but large and carefully selected intervals.
SMARTS adopts the former approach and uses uniform sampling with a large number of very small intervals, and achieves one of the best accuracy/time tradeoff (0.64% CPI error/50 million instructions on the Spec benchmarks and SimpleScalar [4]). However, due to the small size of intervals (around 1,000 instructions), it must continuously warm-up the main SRAM structures in the *functional simulator* (especially the caches, possibly the branch predictors), which is thus called *functional warm-up*. The main limitation of this approach is *practicality* not efficiency: a range of cache mechanisms, for instance prefetching mechanisms, do not lend well to this warm-up approach. Prefetching naturally affects cache behavior, so the prefetching mechanism should be added to the functional simulator as well, but it requires timing information, which the functional simulator does not have. Moreover, with this approach to warm-up, whenever an architecture optimization affects a large processor SRAM structure (cache, branch predictor, TLB,...), it should ideally be implemented in both the functional and timing simulator. This is fairly impractical if not impossible, for some mechanisms, such as for prefetching. It also adds a software engineering burden to the user. It must be noted though that the same authors recently proposed to embed the warm-up information in the simulator modules of their FLEXUS [7] infrastructure in order to reduce that burden. The same authors also proposed TurboSMARTS which obviates functional warm-up by checkpointing micro-architectural state such as the content of SRAM, DRAM and register structures. While this method drastically improves overall simulation time by getting rid of full-program functional simulation, it has similar practicality restrictions related to architecture dependence. The authors show how to partially relax these constraints so that the checkpoints can be reused when some architecture parameters vary, but they acknowledged that the method is difficult to adapt to

some structures, such as modern branch predictors. Still, more recently, Barr et al. [1] have proposed to compress all the trace information required for warming branch predictors; this approach enables to tackle a wider range of branch predictors, though it is still specific to that type of architecture components.

EXPERT [12] adopts the second sampling approach, i.e., wisely picking a few intervals of different sizes; the selection is based on characteristic program constructs, such as loops or subroutines. This program-aware approach to interval selection brings excellent accuracy (0.89% of CPI error and 1 billion[1] simulated instructions on average, again on Spec benchmarks and SimpleScalar). However, because of the small interval granularity/size, it has to resort to continuous warm-up in the functional simulator like SMARTS, with the same practicality limitations. EXPERT can achieve further gains in accuracy and time by pre-characterizing sampling intervals using simulations, but again, this approach raises significant practicality and stability issues when the architecture varies. Also, EXPERT uses checkpointing to drastically reduce simulation time, like TurboSMARTS, but it is again based on micro-architectural state (caches and branch predictors), i.e., architecture-dependent warm-up.

The original SimPoint version [20] was the first step toward wisely picking sampling intervals, based on basic block frequency characteristics. However, having few but large intervals achieved a poorer accuracy/time tradeoff than SMARTS and EXPERT later did (CPI error of 3.7% and 500 million instructions, again with the same benchmarks and simulator). Even though most SimPoint articles assume perfect warm-up (implemented using functional warm-up) [20, 16, 11], the original sampling interval sizes were large enough (100 million instructions) that good accuracy could be achieved without warm-up; and our own SimPoint experiments confirm that warm-up has little impact with such large traces. In the past few years, the SimPoint group has experimented with more but smaller intervals (10 and 1 million instruction intervals and a maximum of 300 simulation points [16, 11]) in order to achieve a better accuracy/time tradeoff, and eventually, they recently proposed a variable-length interval technique [10], in the spirit of EXPERT, but with a different interval selection approach. While this latter technique exhibits again very good accuracy ($\simeq$2.5% of CPI error over 9 Spec benchmarks), the simulation time is still long ($\simeq$1 billion instructions); it also assumes again perfect warm-up before the sampled intervals, which makes it difficult to evaluate the actual accuracy/time/practicality tradeoff.

[1] This number is an approximation derived from the article figures.

In this article, we focus on practicality, and take a holistic approach by considering sampling (trace partitioning), clustering (sample selection), and warm-up together. We make no compromise on practicality by strictly relying on the timing simulator for warm-up, as opposed to the functional simulator or checkpointing. In other words, it is not necessary to implement caches, TLBs, or branch-prediction tables warm-up in the functional simulator, or to have checkpoints include micro-architectural state. We set an accuracy goal on the same order of SMARTS/TurboSMARTS, EXPERT or SimPoint VLI, e.g., on the order of a one or few percent, and we then try to minimize the number of total simulated instructions (warm-up and measurement). Though our number of simulated instructions is higher than for SMARTS/TurboSMARTS (50 million instructions), we have no restrictions on practicality, due to functional simulator/checkpointing warm-up. Moreover, this approach will enable the replacement of functional simulation with micro-architecture-independent checkpointing for further reducing overall simulation time, without the need to resort to micro-architectural state checkpointing as in TurboSMARTS.

We achieve these results through a method which combines several contributions. (1) First, our technique splits the trace into variable-size regions to capture program dynamic control flow [17]. The main benefit of variable size regions is to decompose the program trace into a very large number of regions. Smaller regions are better for sampling accuracy, but they have two caveats: they are more sensitive to warm-up, and they increase the number of regions, straining clustering techniques. Our method avoids both caveats because some of the regions remain large and thus less sensitive to warm-up, and the presence of these large regions keeps the total number of regions reasonable. In some sense, it performs a kind of program-aware "pre-clustering", a bit as if the trace had been split into very small regions and then later clustered into larger ones, but without incurring the high cost of clustering a huge number of very small regions. (2) Because the resulting number of regions is still high, we found that standard clustering techniques [14], as used for fixed-size 1-million intervals in SimPoint, were not appropriate (they become fairly time-consuming). As a result, we have developed a new clustering technique, called IDDCA [18]. Since then, a new version of SimPoint, SimPoint 3.0 [8], has been implemented, which significantly increases the performance of SimPoint; the improvements brought by SimPoint 3.0 are orthogonal, and thus potentially com-

plementary, with our approach. The second motivation for this new clustering technique is to integrate clustering with our budget approach to decide which intervals should receive the largest share of the budget. (3) Finally, we propose a budget-oriented method for distributing the simulation time among both warm-up and performance measurement. The total budget is set by the user according to the maximum accepted simulation time. The budget allocated to a given cluster representative interval (the simulated interval for this cluster) is proportional to the importance of this cluster with respect to the whole trace. And within that cluster budget, the warm-up budget is inversely proportional to the size of the cluster representative interval.

The method achieves an average CPI error of 1.68% and 288 million total simulated instructions per benchmark, over the Spec benchmark suite.

Section 2 presents our method for partitioning the program trace into regions, and how it compares to existing variable-sized sampling intervals methods. Section 3 presents the IDDCA clustering algorithm. Section 4 combines our region partitioning and clustering techniques with a budget-based approach for allocating simulated instructions among warm-up and performance measurement intervals. Section 5 provides an experimental evaluation of our technique.

2 Program Partitioning Into Regions

We describe our approach for partitioning the trace into a very large number of regions. As mentioned above, we use a variable-size approach to have the benefits of both a large number of regions for sampling accuracy, and having several large regions, less sensitive to warm-up.

Recently, EXPERT and SimPoint VLI explored variable-size regions with positive results on accuracy. However, we decided not to retain the EXPERT nor the SimPoint VLI program partitioning methods for the following reasons.

EXPERT partitioning. The principle of EXPERT is to partition the program based on subroutines, with a distinction between long, short and infrequently executed subroutines, then to characterize the performance variability of these subroutines using *simulation*, and then to select the number and location of subroutine representatives. Beyond the practicality issue mentioned in the introduction, and related to continuous warm-up, this partitioning/characterization method has two flaws: the characterization is hardware-dependent and it heavily relies on loops. Hardware-dependent characterization means a code must be simulated on an architecture, before it can be sampled on that architecture. If the target architecture changes, it is hard to anticipate the consequences for the characterization. Loop-based subroutine characterization may also be an issue for codes with complex control flow behavior (multiple `if` statements within a procedure, recursion, etc), and not surprisingly EXPERT demonstrates better results with SpecFp than with SpecInt.

Consider the example of Figure 1 which corresponds to a sequence of basic blocks within the program static control flow graph; each lettered node denotes a basic block. Assuming `BEF` is a large loop called within an even larger loop `ABDCE`, EXPERT would most likely define an interval that encapsulates `BEF` only; multiple invocations of the `BEF` loop would breed multiple intervals.

SimPoint VLI partitioning. SimPoint VLI adopts a different approach, though also based on loops and procedures. Each code structure is numbered and then the trace is viewed as a sequence of identifiers. Then the Sequitur [9, 13] hierarchical pattern matching algorithm is used to identify repeating sequences of variable size within the trace. The main issue with this approach is that it relies on the *exact match* between two sequences within the trace. Programs with complex control flow due to non-trivial `if` statements behavior, may exhibit many different sequences and/or may only enable exact matching for smaller sequences. Therefore, after this pattern matching phase, the SimPoint VLI technique applies several heuristics to relieve this exact match constraint in order to obtain longer sequences.

Consider again the example of Figure 1. Sequitur would partition the sequence into two main recurring parts, i.e., `ABDCE` and `BEF`. As mentioned above, SimPoint VLI adds several heuristics for improving the flexibility of this sequence partitioning.

Region-Based partitioning. Our program partitioning approach is based on the principle that programs can exhibit complex control flow behavior, even within phases. More precisely, the very principle of phases means that programs usually "stay" within a set of static basic blocks for a certain time, then move to another (possibly overlapping) set of basic blocks, and so on. This set of basic blocks can span small code sections such as loops or several subroutines. Moreover, the order and frequency with which these basic blocks are traversed may be very irregular (e.g., `if` statements with very irregular behavior, subroutines which are called infrequently within looping statements, etc...). We call such sets of basic blocks where the program "stays" for a while **regions**. These regions capture the program *stability* while accommodating

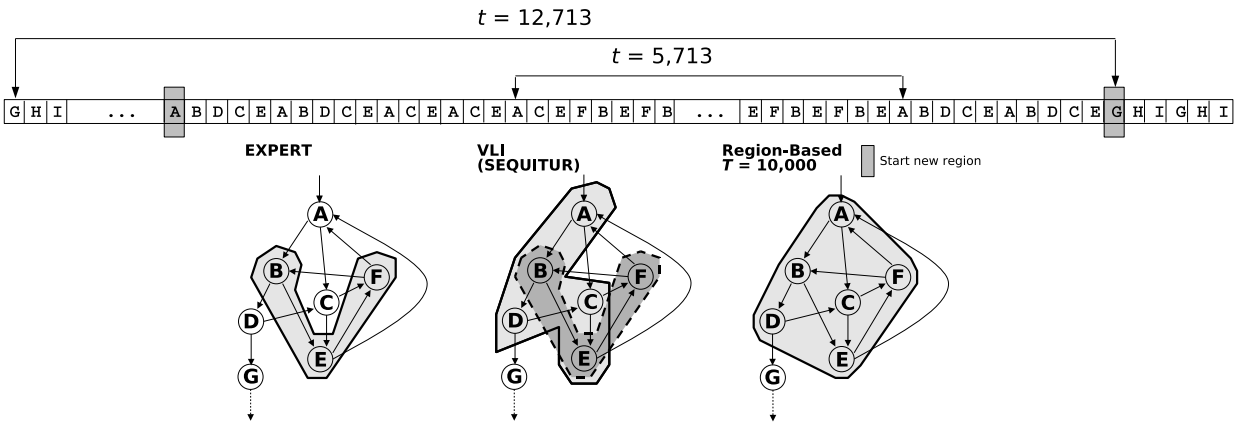

Figure 1: *Program trace partitioning algorithms.*

its *irregular* behavior. We propose a simple method, composed of two rules, for characterizing these basic block regions:

1. Whenever the reuse distance between two occurrences of the same basic block (expressed in number of basic blocks) is greater than a certain time T, the program is said to leave a region.

2. After the program has left a region, application of rule 1 is suspended during T basic blocks, in order to "learn" the new region.

Implicitly, the method progressively builds a pool of basic blocks: whenever a new basic block is accessed, it examines whether this basic block has been recently referenced (less than T ago); if so, it assumes the program is still traversing the same region of basic blocks; if not, it assumes the program is leaving this region. Then, the second rule gives time for the program to build the new pool of basic blocks. Consider again the example of Figure 1. Because `A,B,C,D,E,` and `F` are assumed to be all referenced within a time interval smaller than T, they all belong to the same region, in spite of the sometimes irregular control flow behavior. `G,H,` and `I` mark the beginning of a new region because their reuse distance is greater than T.

Since T determines which reuse distances are captured by regions, a fixed value of T can potentially miss key reuses in certain programs or conversely insufficiently discriminate regions in other programs.[2] We use a benchmark-tolerant method to capture "enough but not too many" reuses. The method sets T for each benchmark such that a fixed percentage P of reuse distances are captured in regions, and we experimentally found $P = 99.6\%$ would capture the appropriate amount of reuse, and thus would result in appropriate values of T for all benchmarks. Table 1 shows T and the regions statistics obtained with $P = 99.6\%$.

SPEC	Number of Instructions	T	Num. Regions	Insn. per Region	Number of Clusters
ammp	326,548,908,728	45,000	183,558	1,778,996	49
applu	223,883,652,707	1,500	187,278	1,195,462	37
apsi	347,924,060,406	3,000	187,311	1,857,450	44
art	41,798,846,919	1,500	112,350	372,041	42
bzip2	108,878,091,744	25,000	170,903	637,075	318
crafty	191,882,991,994	100,000	199,499	961,824	527
eon	80,614,082,807	20,000	194,912	413,592	92
equake	131,518,587,184	2,000	196,991	667,637	17
facerec	211,026,682,877	35,000	196,206	1,075,536	22
fma3d	268,369,311,687	15,000	184,667	1,453,260	73
galgel	409,366,708,209	70,000	111,399	3,674,779	140
gap	269,035,811,516	90,000	192,658	1,396,442	92
gcc	46,917,702,075	20,000	95,529	4,911,357	323
gzip	84,367,396,275	30,000	170,966	493,475	167
lucas	142,398,812,356	100	187,849	758,049	56
mcf	61,867,398,195	25,000	178,469	346,653	54
mesa	281,694,701,214	80,000	187,916	1,499,046	16
mgrid	419,156,005,842	2,500	54,440	7,699,412	32
parser	546,749,947,007	300,000	177,738	3,076,157	507
perlbmk	39,933,232,781	100,000	41,866	953,834	129
sixtrack	470,948,977,898	9,500	183,823	2,561,970	46
swim	225,830,956,489	400	75,740	2,981,660	54
twolf	346,485,090,250	200,000	161,142	2,150,184	28
vortex	118,972,497,867	80,000	190,722	623,806	31
vpr	84,068,782,425	8,500	193,173	435,199	155
wupwise	349,623,848,084	200,000	13,696	25,527,442	16
Average	231,987,140,463	61,130	151,915	2,712,371	118

Table 1: *Region statistics and* T.

For some programs, the average region size is of the order of a few hundred thousands instructions, with some regions as small as a few thousands instructions in `crafty`. Thanks to a mix of large and small regions in each pro-

[2] Note however that we did observe very good average accuracy/time trade-offs for the same T value applied across all benchmarks.

gram, the total number of regions is not excessively high (several tens thousands to a few hundreds thousands). But it is large enough that the k-means [14] clustering method used in SimPoint would take an excessively long time (of the order of one day per benchmark). The IDDCA clustering method presented in the next section can reduce clustering time by more than two orders of magnitude.

3 Clustering a Large Number of Regions Using IDDCA

In this section, we present our technique for clustering a large number of regions. Below, we indifferently use the term "region" or the more classic term "interval". The distance between two intervals is defined as the distance between their two Basic-Block Vectors, as proposed in SimPoint [20].

The popular k-means clustering technique has three main shortcomings: (1) the method works by randomly selecting intervals at the start-up phase, so that several runs of the method on the same trace may not provide the same sampling intervals, and thus the same accuracy results; (2) the number of clusters is a user-selected parameter, but it is sensitive to benchmarks/traces, so that inappropriately setting this parameter can degrade either simulation time or accuracy; (3) the method requires multiple passes which may be impractical for a large number of intervals.

We ran the default SimPoint 2.0 clustering script (`runsimpoint`; default parameters except for *max_k* = 100) on our region partitioning in order to evaluate its execution time. k-means requires 21 hours per benchmark, on average, on an Athlon XP 2800+, (and up to two days for `crafty`), against 9 minutes on average for IDDCA (and 44 minutes for `crafty`), see Figure 2. SimPoint version 3.0 [8] has significantly improved the performance of the clustering time, essentially by reducing the number of intervals in the trace to which clustering is applied, see Figure 3. With this new approach, SimPoint 3.0 is able to perform clustering from 10 to 50 times faster than SimPoint 2.0 while keeping the same accuracy results. Note that the approach used in SimPoint 3.0 is orthogonal to the approach used in IDDCA so we expect IDDCA to benefit as well from intervals reduction, but we have not yet evaluated the combined technique.

IDDCA algorithm. Our clustering method is called IDDCA (*Interleaved Double DCA*), and it is derived from the *Dynamical Clustering Analysis* (DCA) [2] clustering method, and adapted to sampling. IDDCA is an online algorithm, clustering regions one at a time, constantly refining the number of clusters and their centroids. This dynamic process relies upon three different parameters: Θ_{new}, Θ_{merge} and Θ_{step_factor}.

Intuitively, Θ_{new}, Θ_{merge} and Θ_{step_factor} control the creation and merging of clusters. Θ_{new} and Θ_{merge} are threshold distance parameters for respectively determining when a point is far enough from other clusters to induce the creation of a new cluster, or close enough from an existing cluster to be merged into it. Θ_{step_factor} determines the rate at which these threshold distances change. Θ_{new} and Θ_{merge} are initialized using a simple heuristic: 10% of the distance between the global centroid (centroid of all regions) and the farthest region. Θ_{step_factor} is related to the number of data points, but the clustering method is robust enough to tolerate the same Θ_{step_factor} value across all Spec benchmarks, empirically set to $\Theta_{step\_factor} = 10^{-5}$.

IDDCA starts with two elements:

- An empty cluster list;
- And the list of regions to cluster (called R). The regions are regularly *interleaved* in this list, because it makes the online clustering method less sensitive to the original program trace order. Let us assume there are N regions in the trace and the interleaving factor is I, then the list is the following:

  $$1, N/I + 1, 2N/I + 1, \ldots, [(I-1)\times N/I] + 1,$$

  $$2, N/I + 2, 2N/I + 2, \ldots, [(I-1)\times N/I] + 2,$$

  $$\ldots$$

  The method is fairly insensitive to interleaving for $I \geq 2$ and we selected $I = 10$ for all benchmarks. Note that randomly picking regions would have performed similarly well or better, and was not used simply due to implementation constraints.

Then, the IDDCA algorithm is the following one:

1. Pick a region (r) from the list of regions R; if there is no cluster yet, create a first cluster containing region r and go to step 5.
2. Find the cluster (c_i) with the closest centroid to the current region r and compute the distance (d) between r and the centroid of c_i.
3. If d is greater than Θ_{new}, then create a new cluster containing the current region r.
4. If d is less or equal to Θ_{new} then:
   - Add r to cluster c_i and update c_i centroid accordingly.

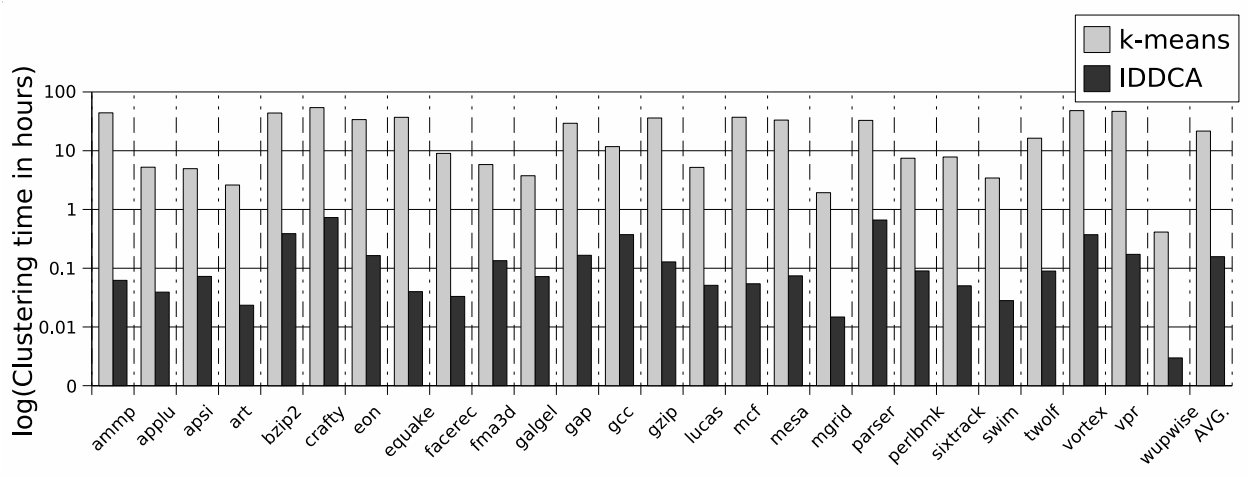

Figure 2: *Clustering time of IDDCA vs. k-means (logarithmic scale).*

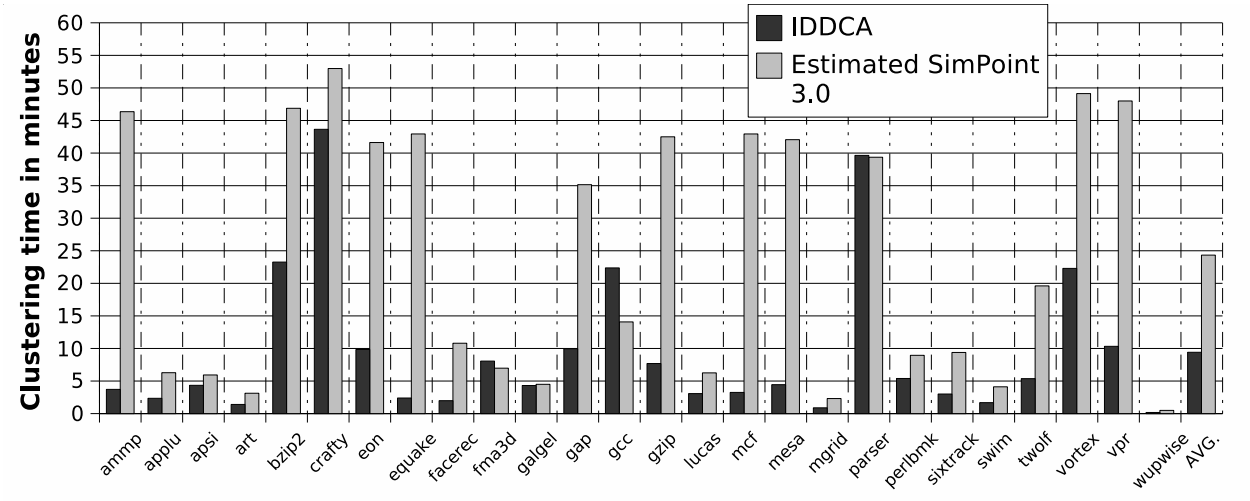

Figure 3: *Clustering time of IDDCA vs. Estimated k-means in SimPoint 3.0.*

- Find the cluster (c_j) with the closest centroid to that of c_i. If the distance between the centroids of c_i and c_j is less or equal to Θ_{merge} then merge the two clusters into a unique one and compute its centroid.
- Update Θ_{new} and Θ_{merge} thresholds so as to make cluster creation and merger more difficult. For that purpose, increase Θ_{new} and decrease Θ_{merge} as follows: $\Theta_{new} = \Theta_{new}/(1-\Theta_{step\_factor})$ and $\Theta_{merge} = \Theta_{merge} \times (1-\Theta_{step\_factor})$.

5. Remove r from the list of regions R. If there are no more regions in R, then the process terminates, otherwise go to step 1.

At the end of this process IDDCA has created a set of clusters. If one of the clusters contains more than 90% of the regions, then IDDCA is hierarchically applied again within this cluster; until clustering is spread enough (no cluster accounts for 90% or more of the regions). Finally, the instructions which must be simulated (sampled) are the region individuals which are the closest regions to the clusters centroids, one per cluster.

Weighted vs. Unweighted IDDCA. Because large regions represent a greater share of the global execution trace than small regions, regions should normally be weighed with their size when computing the centroid. SimPoint VLI is weighing intervals with their size when

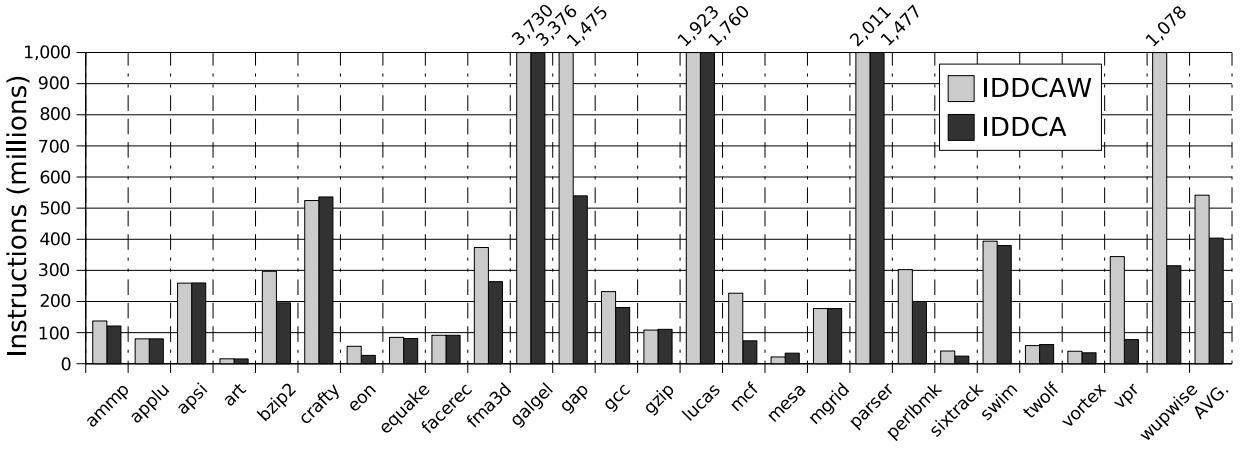

Figure 4: *Simulated instructions with IDDCA and weighted IDDCA (IDDCAW).*

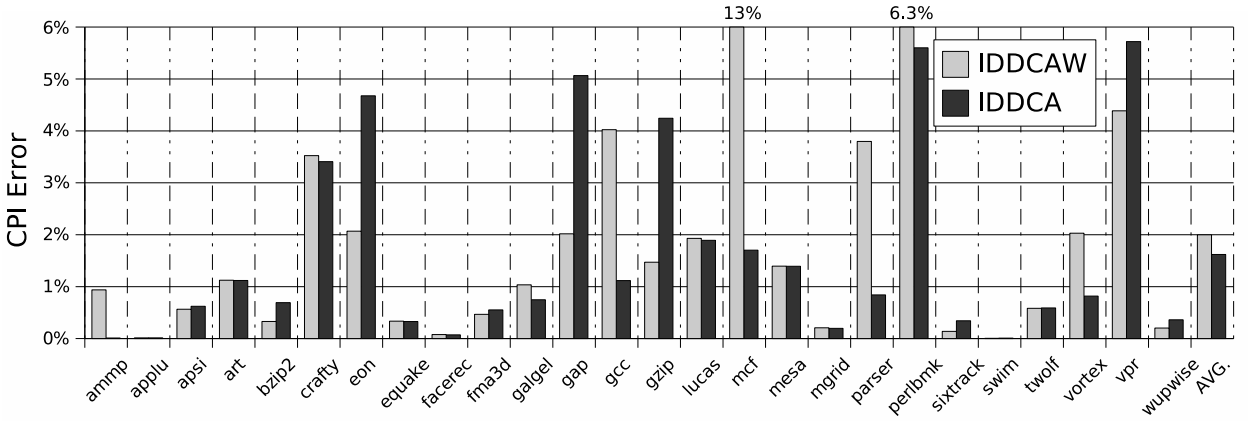

Figure 5: *Simulation accuracy with IDDCA and weighted IDDCA (IDDCAW).*

applying k-means. We decided not to weigh regions with their size in order to privilege a reduction of sampling size over accuracy. This choice was driven by initial sampling results which suggested an effort was required on size rather than accuracy.

Still, in order to investigate the effect of not weighing the clusters, we have run IDDCA clustering with weighted clusters. As expected, the total sampling size increases, and rather signifi cantly (34%), see Figure 4. More surprisingly, the accuracy is lower with weighted clusters than with unweighted clusters: 2.00% CPI error for weighted clusters versus 1.62% for unweighted clusters, see Figure 5. The two observations combined empirically validate the unweighted approach.

The reason why unweighted clustering performs so well is related to both the size distribution of intervals within a cluster and to warming. Within the same cluster, interval size can vary signifi cantly, even within intervals at approximately the same distance from the centroid. As a result, it is possible to sometimes drastically reduce the representative interval size without signifi cantly affecting accuracy. Moreover, the results of Figure 4 already incorporate warming since we want to evaluate the best clustering strategy for our global method. Since the latter method is based on a fi xed budget and distributes the budget between sampling and warming, any simulated instruction budget not consumed by sampling can be spent on warming. As a result, the overall accuracy of the method improves when smaller sampling interval representatives are selected. This latter property illustrates the benefi ts of properly integrating the different components of a sampling strategy.

4 Budget-Oriented Integration of Warm-Up and Sampling

Warm-up is implemented using the main simulator (as opposed to the functional simulator or checkpointing), so warm-up and performance measurement share the same simulation budget. Because simulation is costly, we must take great care to wisely allocate the simulated instructions. And due to both variable-size regions and warm-up implemented through simulation, we must determine what is the budget allocated to each cluster (one region representative is simulated per cluster). The general philosophy is: spend the budget where it's most needed (and in the process, try to minimize the total needed budget).

In that spirit, we make two simple observations. (1) The weight of each cluster should be factored in when allocating its (warm-up and measurement) instruction budget; the cluster weight is defined as the total number of trace (dynamic) instructions in the cluster, which is itself the sum of the lengths of all intervals in the cluster. And (2) the length of each cluster representative region should be factored in when determining the warm-up size for this region.

Let us go back to observation (1). The goal of clustering methods, such as IDDCA or k-means, is to find a representative for each cluster of regions. Not all clusters contain the same total number of instructions; for instance, the cluster sizes range from 57,193 instructions to 430 billion instructions in `sixtrack`. Naturally, when extrapolating performance statistics collected for each cluster representative to the whole program trace, the relative weight of each cluster is factored in. Unlike for the clustering method, this weighing has only an impact on accuracy, not size. Weighing means that the performance measurement of some of the representatives will have a greater impact on the total estimated performance than others. So, we should allocate a greater share of the simulated instruction budget to representatives of large clusters in order to more accurately estimate their performance.

The number of simulated instructions allocated per region consists of the region size plus the additional instructions simulated for warm-up purposes. Which brings observation (2). If a cluster representative region is large (the representative itself, not necessarily the cluster), then it will need less warm-up instructions as the start-up effect will be diluted in the simulation of the cluster representative region. Conversely, small representative regions need significant warm-up, which is a key reason why SMARTS and EXPERT use continuous functional simulator/checkpointing-based warm-up.

Determining measurement and warm-up sizes. Let us call B the total instruction budget, i.e., the maximum number of simulated instructions (including warm-up). Let us number clusters i, with $1 \leq i \leq k$, where k is the total number of clusters, and let us call S_i the total size (in number of instructions) of cluster i; the clusters are ordered by decreasing size, i.e., $S_i > S_j$, if $i < j$. f_i is the weight factor of cluster i over the whole program trace size ($f_i = \frac{S_i}{\sum_{r=1..k} S_r}$), and s_i is the size of the representative region of cluster i.

Because of observation (1), we distribute the budget for each cluster based on the global weight f_i of the cluster. For that purpose, we define B_i as the maximum simulation budget for cluster i (measurement and warm-up); $B_1 = B \times f_1$ and $B_i = (B - \sum_{j=1..i-1} B_j) \times \frac{f_i}{\sum_{l=i..k} f_l}, \forall\, i > 1$, which can be simplified to $B_i = B \times f_i$ if all the clusters are considered, i.e. $\sum_{i=1..k} f_i = 1$. The actual number of simulated instructions for cluster i is $r_i + w_i$ where r_i is the measurement size (it is a subset of the representative of cluster i), and w_i is the warm-up size.

Since the measurement size r_i must be smaller than the budget B_i, i.e., $r_i = \min(s_i, B_i)$, we sometimes need to truncate the simulation of the cluster representative. It rarely degrades accuracy, thanks to the looping behavior which is at the core of our region-partitioning scheme.

Because of observation (2), we preferably allocate warm-up instructions to small samples, within the constraint of budget B_i, i.e., $w_i = B_i - r_i$. The warm-up instructions w_i are instructions preceding the representative of cluster i. Now, due to our region-based partitioning approach, these instructions may reference code sections and data structures which are distinct from the ones referenced in the representative. To avoid simulating useless warm-up instructions, we use the BLRL [5] (*Boundary Line Reuse Latency*) technique for determining the size of the useful warm-up interval. BLRL is an architecture-independent method which consists in collecting the memory addresses and branch instruction addresses used in the sampled interval, and to identify the earliest point in the trace before the interval where they will be all accessed. By starting the warm-up at that point, most SRAM structures are likely to be adequately warmed-up (e.g., the first access to an address will be correctly identified as a hit or a miss) independently of the SRAM structures sizes. However, under that constraint, the actual warm-up interval per sampled region can be very large (e.g., `parser` requires more than 2 billion warm-up instructions for a region of only 1.8 million instructions). For that reason, the authors propose to set a percentage threshold of the sampled interval addresses covered in the warm-up interval, thereby relaxing

Instruction Cache	16K 4-way set-associative, 32 byte blocks, 1 cycle latency
Data Cache	16K 4-way set-associative, 32 byte blocks, 1 cycle latency
L2 Cache	128K 8-way set-associative, 64 byte blocks, 12 cycle latency
Main Memory	120 cycle latency
Branch Predictors	hybrid - 8-bit gshare w/ 2k 2-bit predictors + a 8k bimodal predictor
O-O-O Issue	out-of-order issue of up to 8 operations per cycle, 64 entry re-order buffer
Memory Disambiguation	load/store queue, loads may execute when all prior store addresses are known
Registers	32 integer, 32 fbating point
Functional Units	2-integer ALU, 2-load/store units, 1-FP adder, 1-integer MULT/DIV, 1-FP MULT/DIV
Virtual Memory	8K byte pages, 30 cycle fi xed TLB miss latency after earlier-issued instructions complete

Table 2: *Baseline simulation model.*

the constraint and signifi cantly reducing the warm-up interval size (we used a threshold of 95% across all benchmarks). Still, because our budget approach introduces a size constraint on the warm-up interval, we slightly modify BLRL by limiting the warm-up size to w_i.

5 Evaluation

For evaluation purposes, we used the SimpleScalar [4] 3.0b toolset for the Alpha ISA and experimented with all 26 Spec CPU2000 benchmarks. To create the regions we used the *sim-fast* functional simulator. Table 2 shows the microarchitecture confi guration used for our experiments.

Figures 6 and 7 respectively show the number of instructions and accuracy for our budget approach (setting the budget to $B = 500M$ instructions), and different confi gurations of SimPoint (the maximum number of samples is set to 50 for 10M intervals, and to 100 for 1M intervals, so as to provide a fair accuracy/size comparison). We use perfect warm-up for SimPoint as in most of the articles [20, 16, 11, 10] (recall SimPoint treats sampling as an issue independent from warm-up); we sometimes use no warm-up for comparison purposes. As mentioned in the introduction, while the accuracy of SimPoint 10M is barely sensitive to warm-up, SimPoint 1M becomes fairly sensitive (from 0.7% down to 2.4%), and the trend can only worsen as the sample size decreases. Therefore, while SimPoint 1M requires little instructions compared to SimPoint 10M or our budget approach with warm-up, it would actually require additional instructions for warm-up in order to preserve its accuracy. Our budget approach has lower accuracy but requires fewer instructions than SimPoint 10M. More importantly, our contribution does not lay so much in this instruction budget reduction, but in the fact that the user needs not worry about setting the appropriate sample and warm-up sizes for a new given program, it is all integrated in the partitioning and budgeting approach. All the user needs to set/decide is the maximum simulation budget (i.e., time).

Still, for several cases, especially `eon` and `vpr`, our budget approach performs signifi cantly worse than SimPoint. Some programs have very small but frequently recurring regions, which translates into clusters with many small intervals. And performance is more variable across small intervals, i.e., it is harder to reach steady-state performance after just a few hundred thousands instructions. This is in part due to the higher influence of start-up state on performance for such small intervals. This variability, in turn, can result into signifi cant performance estimation error. As shown in Table 1, `eon` and `vpr` have particularly small regions on average, around 400,000 instructions. While small regions are not necessarily synonymous with performance variability and higher error, they are a potentially aggravating factor. Still, these two codes highlight more the necessity to fi ne-tune our heuristic than a shortcoming, since both codes use only around 20 million instructions for sampling, and 80 million overall, i.e., a small fraction of the total available budget of 500 million instructions.
A possible extension of our method could be to simulate multiple intervals within clusters where the most representative intervals are small, e.g., 10% of the cluster budget. Not only it would average out the performance variability within such clusters, but it would also provide a means for estimating the error within such clusters. The latter would provide a signifi cant enhancement to our technique because one of the shortcomings of clustering-based techniques compared to statistical simulation techniques is that they cannot easily provide a confi dence estimate of the error [16].

Other codes like `gap` and `perlbmk` also behave worse than SimPoint 10M. However, it must be noted that SimPoint 1M without warm-up behaves signifi cantly worse in both cases as well. In fact these codes illustrate the diffi culty of properly selecting both regions and warm-up size. They show that systematic techniques like SimPoint may perform well or poorly depending on how the user selects this interval size, unless the user is willing to engage in a trial and error process for selecting the size. Our budget approach may not be optimal, but it does attempt to shield the user from such decisions by selecting regions sizes automatically.

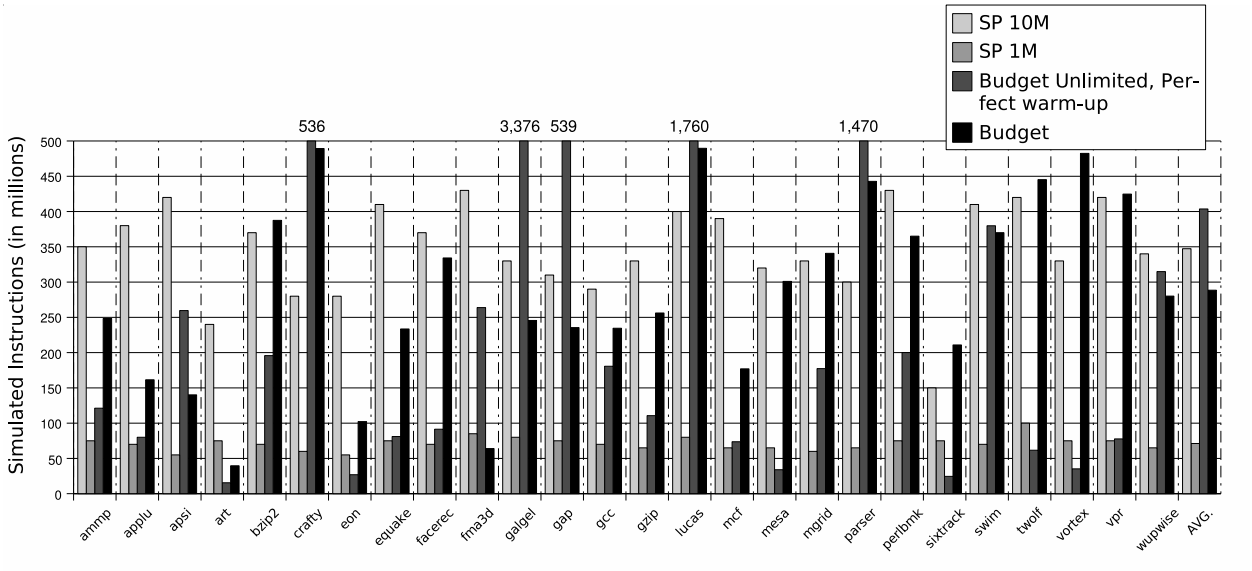

Figure 6: *Number of simulated instructions with different sampling techniques.*

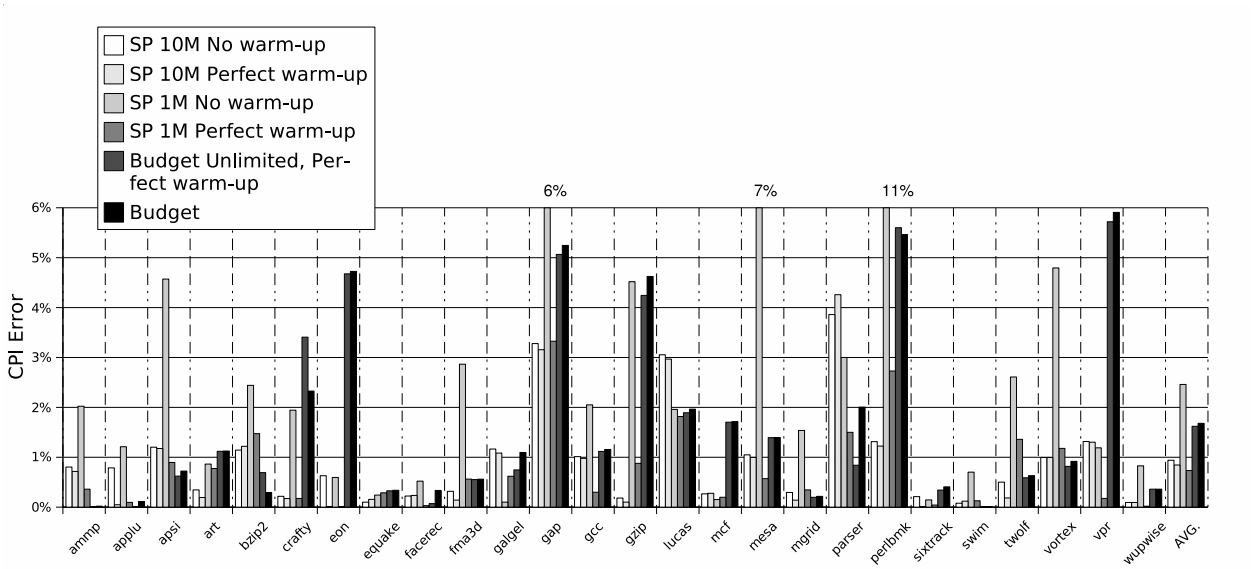

Figure 7: *CPI error.*

We also evaluated our approach without any budget limitation and no budget spent on warm-up, see `Budget Unlimited, Perfect warm-up` (for the Perfect warm-up bars, none of the budget is allocated to warm-up). We can see that wisely allocating the budget allows drastic reductions of the number of simulated instructions in several cases, with limited impact on accuracy (from 1.62% to 1.68%). In some cases, the unlimited budget requires less instructions than the standard budget approach, because the latter includes warm-up. Overall, our allocation strategies result in a total budget which is signifi cantly lower than the maximum accepted budget, set at $B = 500$ million instructions in these experiments.

Figure 8 displays, for each benchmark, how our approach actually distributes its instruction budget between measurement and warm-up. Obviously, the number of simulated instructions devoted to measurement is rather low (only 84 million instructions on average). This value

is close to the number of instructions simulated by SimPoint with 1 million instructions samples (71 million instructions). Warm-up uses most of the instruction budget, with an average of 70% of the total number of simulated instructions. This observation suggests that warm-up and sampling should not be considered separately, especially if the goal is to develop an architecture-independent sampling method by implementing warm-up through simulation.

6 Conclusions

The rationale for our sampling approach is that some of the most recent and efficient sampling techniques have practicality shortcomings due to their warm-up approach (in the functional simulator or by checkpointing micro-architectural state, or simply using perfect warm-up on the principle of separating sampling and warm-up issues). These shortcomings can make it difficult to explore specific and/or a large range of architectural organizations. They are not compatible either with current and upcoming modular simulation frameworks, where the sampling technique will be implemented in the common simulation engine; thus, it will have to be architecture-independent and transparent to the user. This technique will be one of the sampling options implemented in the UniSim framework under development.

Our sampling+warm-up approach focuses on transparency and architecture independence, and still achieves an accuracy/time tradeoff that is of the same order of magnitude as the best sampling techniques. There are three key features in our approach: a novel trace partitioning into variable-size regions which provides a useful compromise between many small intervals and few large intervals, a clustering technique capable of harnessing a large number of intervals, and a budget-oriented distribution of simulated instructions between warm-up and measurement.

References

[1] Kenneth C. Barr and Krste Asanovic. Branch trace compression for snapshot-based simulation. In *International Symposium on Performance Analysis of Systems and Software*, February 2006.

[2] A. Baune, F. T. Sommer, M. Erb, D. Wildgruber, B. Kardatzki, G. Palm, and W. Grodd. Dynamical Cluster Analysis of Cortical fMRI Activation. In *NeuroImage 6(5)*, pages 477 – 489, May 1999.

[3] Michael Van Biesbrouck, Lieven Eeckhout, and Brad Calder. Efficient Sampling Startup for Sampled Processor Simulation. *International Conference on High Performance Embedded Architectures & Compilers*, 2005.

[4] Doug Burger, Todd M. Austin, and Steve Bennett. Evaluating Future Microprocessors: The SimpleScalar Tool Set. Technical Report CS-TR-1996-1308, University of Wisconsin-Madison, Madison, 1996.

[5] L. Eeckhout, S. Eyerman, B. Callens, and K. De Bosschere. Accurately Warmed-up Trace Samples for the Evaluation of Cache Memories. In I. Banicescu, editor, *Proceedings of the High Performance Computing Symposium - HPC2003*, pages 267–274, Orlando, FL, USA, 4 2003. SCS.

[6] Joel S. Emer, Pritpal Ahuja, Eric Borch, Artur Klauser, Chi-Keung Luk, Srilatha Manne, Shubhendu S. Mukherjee, Harish Patil, Steven Wallace, Nathan L. Binkert, Roger Espasa, and Toni Juan. Asim: A performance model framework. *IEEE Computer*, 35(2):68–76, 2002.

[7] Flexus. http://www.ece.cmu.edu/ simflex/flexus.html.

[8] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. Simpoint 3.0: Faster and More Flexible Program Analysis. *MOBS '05: Workshop on Modeling, Benchmarking and Simulation*, june 2004.

[9] James R. Larus. Whole program paths. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 259–269. ACM Press, 1999.

[10] Jeremy Lau, Erez Perelman, Greg Hamerly, Timothy Sherwood, and Brad Calder. Motivation for Variable Length Intervals and Hierarchical Phase Behavior. *ISPASS '05: IEEE International Symposium on Performance Analysis of Systems and Software*, 2005.

[11] Jeremy Lau, Stefan Schoenmackers, and Brad Calder. Structures for Phase Classification. *ISPASS '04: IEEE International Symposium on Performance Analysis of Systems and Software*, 2004.

[12] Wei Liu and Michael C. Huang. EXPERT: expedited simulation exploiting program behavior repetition. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 126–135. ACM Press, 2004.

[13] C. G. Nevill-Manning and I. H. Witten. Compression and Explanation Using Hierarchical Grammars. *The Computer Journal*, 40(2/3):103–116, 1997.

[14] Dan Pelleg and Andrew W. Moore. X-means: Extending K-means with Efficient Estimation of the Number of Clusters. In *ICML '00: Proceedings of the Seventeenth International Conference on Machine Learning*, pages 727–734. Morgan Kaufmann Publishers Inc., 2000.

[15] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. Using SimPoint for accurate and efficient simulation. *SIGMETRICS Perform. Eval. Rev.*, 31(1):318–319, 2003.

[16] Erez Perelman, Greg Hamerly, and Brad Calder. Picking Statistically Valid and Early Simulation Points. In *PACT '03: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, page 244. IEEE Computer Society, 2003.

[17] Daniel Gracia Pérez, Hugues Berry, and Olivier Temam. Budgeted Region Sampling (BeeRS): Do Not Separate Sampling From Warm-Up, And Then Spend Wisely Your

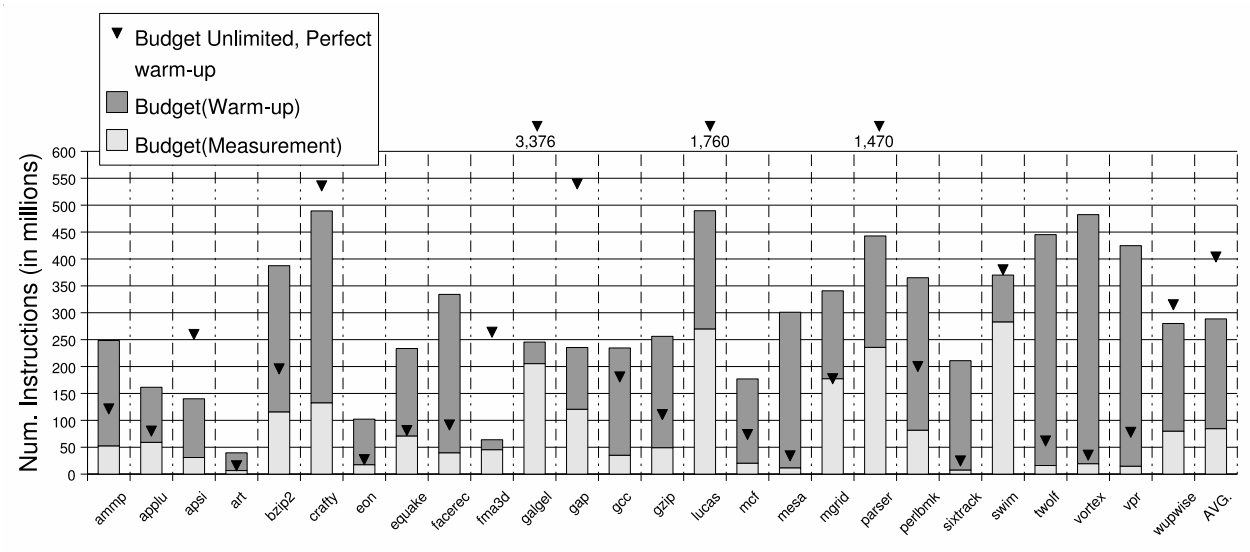

Figure 8: *Distribution of the number of measurement and warmed-up instructions with the budget approach.*

Simulation Budget (6-pages abstract). *ISSPIT 5: IEEE International Symposium on Signal Processing and Information Technology*, December 2005.

[18] Daniel Gracia Pérez, Hugues Berry, and Olivier Temam. IDDCA: A New Clustering Approach For Sampling. In *MoBS '05: Workshop on Modeling, Benchmarking and Simulation*, 2005.

[19] Daniel Gracia Pérez, Gilles Mouchard, and Olivier Temam. MicroLib: A Case for the Quantitative Comparison of Micro-Architecture Mechanisms. In *MICRO-37: Proceedings of the 37th International Symposium on Microarchitecture*, pages 43–54. IEEE Computer Society, 2004.

[20] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. *SIGOPS Oper. Syst. Rev.*, 36(5):45–57, 2002.

[21] Systemc v2.0.1 language reference manual, 2003. http://www.systemc.org/.

[22] UNISIM: UNIted SIMulation environment. http://unisim.org.

[23] Manish Vachharajani, Neil Vachharajani, David A. Penry, Jason A. Blome, and David I. August. Microarchitectural Exploration with Liberty. In *the 34th Annual International Symposium on Microarchitecture*, Austin, Texas, USA., December 2001.

[24] Thomas F. Wenisch, Roland E. Wunderlich, Babak Falsafi, and James C. Hoe. TurboSMARTS: Accurate Microarchitecture Simulation Sampling in Minutes. *SIGMETRICS '05*, June 2005.

[25] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 84–97. ACM Press, 2003.